UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1301

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

A LOGIC PROGRAM FOR TRANSFORMING SEQUENT PROOFS TO NATURAL DEDUCTION PROOFS

Amy FELTY

Octobre 1990

RR_1301

A Logic Program for Transforming Sequent Proofs to Natural Deduction Proofs

Amy Felty
INRIA Sophia-Antipolis

Abstract In this paper, we show that an intuitionistic logic with second-order function quantification, called hh^2 here, can serve as a meta-language to directly and naturally specify both sequent calculi and natural deduction inference systems for first-order logic. For the intuitionistic subset of first-order logic, we present a set of hh^2 formulas which simultaneously specifies both kinds of inference systems and provides a direct and concise account of the correspondence between cut-free sequential proofs and normal natural deduction proofs. The logic of hh^2 can be implemented using such logic programming techniques as providing operational interpretations to the connectives and implementing unification on λ-terms. With respect to such an interpreter, our specification provides a description of how to convert a proof in one system to a proof in the other. The operation of converting a sequent proof to a natural deduction proof is functional in the sense that there is always one natural deduction proof corresponding to each sequent proof. Our specification, in fact, provides a direct implementation of the transformation in this direction.

Un Programme Logique pour Transformer des Preuves dans un Calcul des Séquents en Preuves en Déduction Naturelle

Résumé Dans cet article, nous montrons qu'une logique intuitionniste avec quantification du deuxième ordre sur les fonctions, appelée ici hh^2, peut servir comme méta-langage pour la spécification directe et naturelle d'un calcul des séquents et d'un système de déduction naturelle pour la logique du premier ordre. Pour la logique intuitionniste de premier ordre, nous présentons un ensemble de formules hh^2 qui spécifie simultanément les deux sortes de systèmes d'inférence et fournit une explication directe et concise de la correspondance entre les preuves sans coupure dans le calcul des séquents et les preuves normales en déduction naturelle. La logique hh^2 peut être implémentée en utilisant des techniques de programmation logique comme par exemple donner des interprétations opérationelles aux connectives et implémenter l'unification sur les λ-termes. Par rapport à un interpréteur, notre spécification donne une description de la manière de convertir une preuve dans un système en une preuve dans l'autre. L'opération de convertir une preuve dans le calcul des séquents en une preuve en déduction naturelle est fonctionnelle dans le sens qu'il y a toujours exactement une preuve en déduction naturelle qui correspond à chaque preuve dans le calcul des séquents. En fait, notre spécification fournit une implémentation directe de la transformation dans ce sens.

A Logic Program for Transforming Sequent Proofs to Natural Deduction Proofs *

Amy Felty

INRIA Sophia-Antipolis
2004, Route des Lucioles
06565 Valbonne Cedex, France

Abstract

In this paper, we show that an intuitionistic logic with second-order function quantification, called hh^2 here, can serve as a meta-language to directly and naturally specify both sequent calculi and natural deduction inference systems for first-order logic. For the intuitionistic subset of first-order logic, we present a set of hh^2 formulas which simultaneously specifies both kinds of inference systems and provides a direct and concise account of the correspondence between cut-free sequential proofs and normal natural deduction proofs. The logic of hh^2 can be implemented using such logic programming techniques as providing operational interpretations to the connectives and implementing unification on λ-terms. With respect to such an interpreter, our specification provides a description of how to convert a proof in one system to a proof in the other. The operation of converting a sequent proof to a natural deduction proof is functional in the sense that there is always one natural deduction proof corresponding to each sequent proof. Our specification, in fact, provides a direct implementation of the transformation in this direction.

1 Introduction

Intuitionistic logic with quantification at all function types has been proposed as a meta-language for the direct and natural specification of a wide range of logics [5, 12]. As illustrated by these papers, the second-order subset of this logic, called hh^2 here, is all that is required to suitably specify both sequent calculi and natural deduction for first-order logic. From these specifications, it is possible to obtain a description of goal-directed theorem proving and proof checking for these inference systems by providing simple operational interpretations for the connectives of hh^2.

The correspondence between cut-free sequential proofs and normal natural deduction proofs for first-order intuitionistic logic has been formalized in [16, 13].

* To appear in the proceedings of the December 1989 Workshop on Extensions of Logic Programming, Springer-Verlag LNCS series, edited by Peter Schroeder-Heister.

(There, the formal relation between cut-elimination and proof normalization is also explored in detail.) In this paper, we show that it is possible to merge an hh^2 specification of a sequent system for first-order intuitionistic logic with one for natural deduction, resulting in a specification which demonstrates the correspondence between these two systems in a concise and natural manner. The operational interpretations of the connectives of hh^2 provide descriptions for simultaneous proof construction and proof checking in these two inference systems, as well as a description of how to convert a proof in one system to a proof in the other.

The transformation from sequent proofs to natural deduction proofs is "functional" since the relation between sequent proofs and natural deduction proofs is many to one, *i.e.*, there will always be exactly one normal natural deduction proof corresponding to each cut-free sequent proof. Our specification in fact provides a direct implementation of the transformation in this direction with respect to a deterministic logic programming interpreter which implements hh^2. Our specification cannot, however, serve directly as a program for the transformation in the other direction. More control information must be added in order to obtain a program that can transform a natural deduction proof to one of the possibly many sequent proofs to which it is related.

The logic of hh^2 is a sublanguage of *higher-order hereditary Harrop formulas* which serve as the foundation of the logic programming language λProlog [11]. In particular, it is the sublanguage with function quantification restricted to second-order and with no predicate quantification. Thus, an implementation of λProlog includes an interpreter for hh^2. The transformation from sequent proofs to natural deduction proofs has in fact been implemented and tested in λProlog.

In Sect. 2 we present the meta-logic hh^2 and an interpreter for it. In Sect. 3 we discuss the specification of a sequent system for first-order intuitionistic logic using this meta-logic. In Sect. 4 we will see that there are many options in specifying natural deduction. We first present a direct specification of the inference rules and then discuss alternatives, leading to a specification in which only normal proofs can be constructed and which can be merged directly with the sequent specification. Then, in Sect. 5 we demonstrate how these two specifications can be merged, and finally, in Sect. 6 we conclude.

2 The Meta-Logic and Language

The logic of hh^2 extends positive Horn clauses in essentially two ways. First, it replaces first-order terms with the more expressive simply-typed λ-terms. Second, it permits richer logical expressions in both goals and the bodies of program clauses. In particular, implication and second-order universal quantification over functions are permitted.

The types and terms of this language are essentially those of the simple theory of types [1]. We assume a fixed set of *primitive types*, which includes at least the symbol o, the type for propositions. *Function types* are constructed

using the binary infix symbol $\rightarrow$; if τ and σ are types, then so is $\tau \rightarrow \sigma$. The type constructor $\rightarrow$ associates to the right. The *order* of a primitive type is 0 while the order of a function type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where τ_0 is primitive is one greater than the maximum order of $\tau_1, \ldots, \tau_n$.

For each type τ, we assume that there are denumerably many constants and variables of that type. A *signature* is a finite set of constants, such that for each constant c, the type of c has the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $n \geq 0$, τ_0 is primitive, and $\tau_1, \ldots, \tau_n$ do not contain o. If τ_0 is o, then c is a *predicate constant.* In this paper, the types of constants in signatures will always be of order 2 or less.

Simply typed λ-terms are built in the usual way using constants, variables, applications, and abstractions. Equality between λ-terms is taken to mean $\beta\eta$-convertible. We shall assume that the reader is familiar with the usual notions and properties of substitution and α, β, and η conversion for the simply typed λ-calculus. See [7] for a fuller discussion of these basic properties.

The logical connectives are defined by introducing suitable constants as in [1]. The constants $\wedge$ (conjunction) and $\supset$ (implication) are both of type $o \rightarrow o \rightarrow o$, and $\forall_\tau$ (universal quantification) is of type $(\tau \rightarrow o) \rightarrow o$, for all types τ not containing o. The expression $\forall_\tau(\lambda z\ t)$ is written simply as $\forall_\tau z\ t$. An *atomic formula* is of the form $(pt_1 \ldots t_n)$, where $n \geq 0$, p is a predicate constant of the type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow o$, and $t_1, \ldots, t_n$ are terms of the types $\tau_1, \ldots, \tau_n$, respectively. We write A to denote a syntactic variable for atomic formulas. We define two classes of *non-atomic formulas* called *goal formulas* and *definite clauses.* Let G be a syntactic variable for goal formulas and let D be a syntactic variable for definite clauses. These two classes are defined by the following mutual recursion.

$$G := A \mid G_1 \wedge G_2 \mid D \supset G \mid \forall_\tau x\ G$$

$$D := A \mid G \supset A \mid \forall_\tau x\ D$$

Here τ is a type of order 0 or 1 not containing o. We say that a formula has order n if in each of its subformulas of the form $\forall_\tau z\ t$, type τ has order strictly less than n. Thus all goal formulas and definite clauses are second-order. A *logic program* or just simply a *program* is a finite set, generally written as $\mathcal{P}$, of closed definite clauses. In definite clauses of the form $\forall_{\tau_1} x_1 \ldots \forall_{\tau_n} x_n\ (G_1 \wedge \cdots \wedge G_n \supset A)$, we call A the *head* and $G_1 \wedge \cdots \wedge G_n$ the *body.* The subformulas $G_1, \ldots, G_n$ are also called *subgoals.*

Based on properties shown for the full logic of higher-order hereditary Harrop formulas in [9], a sound and complete (with respect to intuitionistic logic) *non-deterministic* search procedure can be described using the following four search primitives.

AND: $G_1 \wedge G_2$ is provable from $\mathcal{P}$ if and only if both G_1 and G_2 are provable from $\mathcal{P}$.

GENERIC: $\forall_\tau x G$ is provable from $\mathcal{P}$ if and only if $[c/x]G$ is provable from $\mathcal{P}$ for some new constant c of type τ not in $\mathcal{P}$ or G.

AUGMENT: $D \supset G$ is provable from $\mathcal{P}$ if and only if G is provable from $\mathcal{P} \cup \{D\}$.

BACKCHAIN: The atomic formula A is provable from $\mathcal{P}$ if and only if there is either (1) a universal instance of a definite clause in $\mathcal{P}$ that is $\beta\eta$-convertible to A, or (2) a universal instance of the form $G \supset A$ such that G is provable from $\mathcal{P}$.

In order to implement a *deterministic* interpreter, it is necessary to make choices left unspecified in the high-level description above. We will make choices as in the λProlog language, many of which are similar to those made in Prolog. For example, logic variables are employed in the BACKCHAIN operation to create universal instances of definite clauses. As a result, second-order unification is necessary since logic variables for functions (with types of order one) can occur inside λ-terms. Also the equality of terms is not a simple syntactic check but a more complex check of $\beta\eta$-conversion. Second-order unification is not in general decidable. In λProlog, full higher-order unification is required and this issue is addressed by implementing a depth-first version of the unification search procedure described in [8]. (See [10, 9].) In this paper, the second-order unification problems that result from programs we present are all rather simple: it is easy to see, for example, that all such problems are decidable.

In the AUGMENT search operation, clauses get added to the program dynamically. Note that as a result, clauses may in fact contain logic variables. The GENERIC operation must be implemented so that the new constant c introduced for x, must not appear in the terms eventually instantiated for logic variables free in the goal or in the program when c is introduced.

Finally, we assume conjuncts are attempted in the order they are presented, and definite clauses are backchained over in the order they are listed in $\mathcal{P}$ using a depth-first search paradigm to handle failures.

In presenting example hh^2 formulas in this paper, we will adopt the syntax of the eLP [3] implementation of λProlog. Fortunately, very little of this syntax is needed here. In particular, a signature member, say f of type $a \to b \to c$, is represented as simply the line:

```
type   f        a -> b -> c.
```

Tokens with initial capital letters will denote either bound or free variables. All other tokens will denote constants. λ-abstraction is written using backslash `\` as an infix operator. Universal quantification is written using the constant `pi` in conjunction with a λ-abstraction, *e.g.*, `pi X\` represents universal quantification over variable `X`. We omit type subscripts for this quantifier. They can always be inferred from context. The symbols `,` and `=>` represent $\wedge$ and $\supset$, respectively. The symbol `:-` denotes the converse of `=>` and is used to write the top-level implication in definite clauses. We omit universal quantifiers at the top level in definite clauses, and assume implicit quantification over all free variables.

Consider a set of primitive types that includes the types `a` and `a_1`. Here,

a_l is meant to represent the type for lists of elements of type a. Consider a signature which includes the following declarations:

```
type    nil_a   a_l.
type    ::_a    a -> a_l -> a_l.
type    memb_a  a -> a_l -> o.
```

where nil_a represents the empty list of elements of type a, ::_a is the cons operator for a-lists, and memb_a is a predicate which takes as arguments an element and a list of elements of type a. We will write ::_a using infix notation. The following hh^2 definite clauses axiomatize the membership relation for type a.

```
memb X (X ::_a L).
memb X (Y ::_a L) :- memb X L.
```

With respect to the deterministic interpreter described above, these formulas implement the standard program testing list membership as often written in Prolog. In this case, the program is restricted to lists of type a. In the programs in this paper, we will make use of such a membership predicate at two primitive types. We will omit type subscripts from the constants nil, ::, and memb since it will be clear from context which of the two is meant. In λProlog, these constants can be treated in a polymorphic fashion similar to their treatment in ML.

3 Specifying Sequent Calculi

Since we will be specifying first-order logic within the logic of hh^2, to avoid confusion we will refer to first-order logic as the *object-logic* to distinguish it from hh^2, the *meta-logic*. To represent first-order logic in hh^2, we introduce two primitive types: form for object-level formulas and tm for first-order terms. The new type form serves to distinguish formulas of the object-logic from formulas of the meta-logic (which have type o). Given these new primitive types, we introduce the following constants for the object-level connectives.

```
type    and     form -> form -> form.
type    or      form -> form -> form.
type    imp     form -> form -> form.
type    neg     form -> form.
type    forall  (tm -> form) -> form.
type    exists  (tm -> form) -> form.
type    false   form.
```

The constant and, for example, takes two formulas as arguments and constructs their conjunction. For readability, we write and, or, and imp using infix notation. By declaring forall and exists to take functional arguments, we have defined object-level binding of variables by quantifiers in terms of lambda abstraction, the meta-level binding operator, as is done in [1]. Thus, bound variables of the object-language are identified with bound variables of type tm at the meta-level. Meta-constants representing object-level constants, function symbols, propositions, and predicates can also be introduced and given appropriate types. For

example, a constant p of type `tm -> form` represents an object-level unary predicate. Using these definitions, the first-order formula $\forall x \exists y(px \supset py)$, for instance, is represented by the λ-term below where `X` and `Y` are meta-variables of type `tm`.

```
(forall X\ (exists Y\ ((p X) imp (p Y))))
```

Figure 1 contains a complete set of inference rules for a sequent calculus for first-order intuitionistic logic. This inference system, which we call L_I, is a variant of the L system in [2]. In this system a sequent is written $\Gamma \longrightarrow A$ where Γ is a *set* of formulas, and A is a formula. Γ is called the *antecedent* of the sequent and A the *succedent*. Following convention, we write A, Γ to denote the set $\Gamma \cup \{A\}$. An *initial* sequent has the form $\Gamma \longrightarrow A$ where $A \in \Gamma$. The

$$\frac{\Gamma \longrightarrow A \qquad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B}\ \wedge\text{-R} \qquad \frac{A, B, \Gamma \longrightarrow C}{A \wedge B, \Gamma \longrightarrow C}\ \wedge\text{-L}$$

$$\frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B}\ \vee\text{-R} \qquad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B}\ \vee\text{-R} \qquad \frac{A, \Gamma \longrightarrow C \qquad B, \Gamma \longrightarrow C}{A \vee B, \Gamma \longrightarrow C}\ \vee\text{-L}$$

$$\frac{A, \Gamma \longrightarrow B}{\Gamma \longrightarrow A \supset B}\ \supset\text{-R} \qquad \frac{\Gamma \longrightarrow A \qquad B, \Gamma \longrightarrow C}{A \supset B, \Gamma \longrightarrow C}\ \supset\text{-L}$$

$$\frac{A, \Gamma \longrightarrow \bot}{\Gamma \longrightarrow \neg A}\ \neg\text{-R} \qquad \frac{\Gamma \longrightarrow A}{\neg A, \Gamma \longrightarrow \bot}\ \neg\text{-L}$$

$$\frac{\Gamma \longrightarrow [y/x]A}{\Gamma \longrightarrow \forall x A}\ \forall\text{-R} \qquad \frac{[t/x]A, \Gamma \longrightarrow C}{\forall x A, \Gamma \longrightarrow C}\ \forall\text{-L}$$

$$\frac{\Gamma \longrightarrow [t/x]A}{\Gamma \longrightarrow \exists x A}\ \exists\text{-R} \qquad \frac{[y/x]A, \Gamma \longrightarrow C}{\exists x A, \Gamma \longrightarrow C}\ \exists\text{-L}$$

$$\frac{\Gamma \longrightarrow \bot}{\Gamma \longrightarrow A}\ \bot\text{-R} \qquad \frac{\Gamma \longrightarrow A \qquad A, \Gamma \longrightarrow C}{\Gamma \longrightarrow C}\ \text{cut}$$

The $\forall$-R and $\exists$-L rules have the proviso that the variable y cannot appear free in the lower sequent.

Fig. 1. The L_I sequent calculus for first-order intuitionistic logic

L_I inference system differs from the sequent system originally given in [6] in that we consider sets instead of sequences of formulas as antecedents and allow a more general form of initial sequent. As a result, in our formulation there is no need for explicit structural rules in the antecedent. The $\bot$-R rule is as in the specification of sequent systems in [15], and corresponds to the usual rule for thinning on the right.

To represent sets of formulas in antecedents, we will use lists of elements of type `form` where the order and number of copies of each element is not significant. Thus we introduce the primitive type `form_l`. In addition, we introduce the primitive type `seq` for sequents and add the signature item `-->` of type `form_l -> form -> seq`. We will write `-->` as an infix operator whose antecedent is a list of formulas and succedent is a single formula.

Finally, we introduce the primitive type `lprf` for the type of sequent proofs. The basic relation between a sequent and its proofs will be represented as a binary predicate at the meta-level by the infix constant `>-` of type `lprf -> seq -> o`. Each inference rule of the sequent calculus will be expressed as a simple declarative fact about this relation. Operationally, with respect to the interpreter described in the previous section, `>-` can be viewed as the theorem proving predicate.

For illustration purposes, we present the specification of a small subset of the inference rules of L_I. The others can be specified similarly. In specifying these rules, there are often many choices in choosing a representation of sequent proofs. In this presentation, we simply choose one. For a more thorough discussion of the issues involved in specifying inference rules and proofs for this inference system and others in hh^2, see [5, 4].

First, consider the $\wedge$-R inference rule in Fig. 1 which introduces a conjunction on the right side of the sequent. This rule has a natural rendering as the following definite clause.

```
(and_r Q1 Q2) >- (Gamma --> (A and B)) :- Q1 >- (Gamma --> A),
                                          Q2 >- (Gamma --> B).
```

This formula may be read as: if `Q1` is a proof of `(Gamma --> A)` and `Q2` is a proof of `(Gamma --> B)`, then `(and_r Q1 Q2)` is a proof of `(Gamma --> (A and B))`. The constant `and_r` is a "proof constructor" which takes two proofs as arguments (the premises of the $\wedge$-R rule) and builds a new proof (its conclusion). Its type is `lprf -> lprf -> lprf`.

The following clause specifies the $\supset$-L rule and illustrates that introductions of logical constants into the antecedent of a sequent can be achieved similarly.

```
(imp_l (A imp B) Q1 Q2) >- (Gamma --> C) :- memb (A imp B) Gamma,
                                            Q1 >- (Gamma --> A),
                                            Q2 >- ((B::Gamma) --> C).
```

The main difference here is that the antecedent is a list instead of a single formula. We use the `memb` predicate (on type `form`) defined in Sect. 2 to choose an implication `(A imp B)` from the list `Gamma`. Note that the proof term in the head of the clause contains a record of the particular implication from `Gamma` to which the rule is applied.

We now consider the quantifier introduction rules. The declarative reading of the $\exists$-R inference rule is captured by the following definite clause.

```
(exists_r T Q) >- (Gamma --> (exists A)) :- Q >- (Gamma --> (A T)).
```

The existential formula of the conclusion of this rule is written `(exists A)` where the variable `A` has functional type `tm -> form`. Thus `A` is an abstraction over first-order terms and `(A T)` represents the formula that is obtained by substituting `T` for the bound variable in `A`. Declaratively, this clause reads: for first-order term `T`, if `Q` is a proof of `(Gamma --> (A T))`, then `(exists_r T Q)` is a proof of `(Gamma --> (exists A))`. Note that the proof term contains a

record of the substitution term T. Operationally, the existential instance (A T) is obtained via the interpreter's operation of β-reduction. Thus β-conversion at the meta-level is used to specify substitution at the object-level.

Next, we consider the $\forall$-R rule, which has the additional proviso that y is not free in Γ or $\forall x A$. This proviso is handled by using a universal quantifier at the meta-level as in the following definite clause.

```
(forall_r Q) >- (Gamma --> (forall A)) :-
   pi Y\ ((Q Y) >- (Gamma --> (A Y))).
```

As in the previous clause, A has functional type. In this case, so does Q; it has type tm -> lprf, and thus the type of the constant forall_r is (tm -> lprf) -> lprf. Declaratively, this clause reads: if we have a function Q that maps arbitrary terms Y to proofs (Q Y) of the sequent (Gamma --> (A Y)), then (forall_r Q) is a proof of (Gamma --> (forall A)). Operationally, the GENERIC search operation is used to insert a new constant of type tm into the sequent. Since that constant will not be permitted to appear in Gamma or A the proviso will be satisfied.

As a final example, consider the cut rule which has the straightforward specification as the following formula.

```
(cut A Q1 Q2) >- (Gamma --> C) :- Q1 >- (Gamma --> A),
                                  Q2 >- ((A::Gamma) --> C).
```

A complete set of hh^2 formulas specifying the rules of L_I serves as a specification of a theorem prover for this inference system. To prove sequent (Gamma --> A), we start with a goal of the form (Q >- (Gamma --> A)) where Q is a logic variable to be filled in with a term representing a proof of (Gamma --> A). Note, though, that there may be multiple definite clauses that can be used in backchaining to prove a particular sequent, or a single clause that can be used repeatedly. For example, if there is an implication in Gamma, it will always be possible to backchain on the clause specifying the $\supset$-L rule. Thus, this set of formulas cannot serve as an implementation with respect to a depth-first interpreter that backchains over definite clauses in a particular order. (See [5] for more on implementing theorem provers and controlling search in this setting.) On the other hand, when Q is a closed term in the original goal, this program behaves as a proof checker and is complete even with respect to the deterministic interpreter. The top-level constant of a proof term completely determines the unique definite clause to be used in backchaining at each step.

By Gentzen's cut-elimination result [6], L_I without the cut rule is a complete set of rules for first-order intuitionistic logic. Thus, by simply eliminating the formula for the cut rule, we obtain a set of hh^2 formulas which serves as a specification for both building and proof checking cut-free proofs.

4 Specifying Natural Deduction

Figure 2 presents the inference rules for natural deduction in intuitionistic logic as presented in [15], called N_I here. The premise of an elimination rule (E-

$$\frac{A \quad B}{A \wedge B}\ \wedge\text{-I} \qquad \frac{A \wedge B}{A}\ \wedge\text{-E} \qquad \frac{A \wedge B}{B}\ \wedge\text{-E}$$

$$\frac{A}{A \vee B}\ \vee\text{-I} \qquad \frac{B}{A \vee B}\ \vee\text{-I} \qquad \frac{A \vee B \quad \begin{matrix}(A)\\ C\end{matrix} \quad \begin{matrix}(B)\\ C\end{matrix}}{C}\ \vee\text{-E}$$

$$\frac{\begin{matrix}(A)\\ B\end{matrix}}{A \supset B}\ \supset\text{-I} \qquad \frac{A \quad A \supset B}{B}\ \supset\text{-E}$$

$$\frac{\begin{matrix}(A)\\ \bot\end{matrix}}{\neg A}\ \neg\text{-I} \qquad \frac{A \quad \neg A}{\bot}\ \neg\text{-E}$$

$$\frac{[y/x]A}{\forall x A}\ \forall\text{-I} \qquad \frac{\forall x A}{[t/x]A}\ \forall\text{-E}$$

$$\frac{[t/x]A}{\exists x A}\ \exists\text{-I} \qquad \frac{\exists x A \quad \begin{matrix}([y/x]A)\\ B\end{matrix}}{B}\ \exists\text{-E}$$

$$\frac{\bot}{A}\ \bot_I$$

The $\forall$-I rule has the proviso that the variable y cannot appear free in $\forall x A$, or in any assumption on which $[y/x]A$ depends.

The $\exists$-E rule has the proviso that the variable y cannot appear free in $\exists x A$, in B, or in any assumption on which the upper occurrence of B depends.

Fig. 2. The N_I natural deduction inference system for first-order intuitionistic logic

rule) containing the connective for which the rule is named is called the *major premise*, and other premises are *minor premises*. The *discharge* of assumptions is indicated by parentheses. For example, in the $\supset$-I rule, (A) indicates that occurrences of A at the leaves are discharged by the application of this rule. A formula occurrence B in a tree is said to *depend* on an assumption A if A occurs as a leaf and is not discharged by a rule application above B. A tree with root B constructed using these inference rules is a *deduction* of B from the set of formulas Γ if all assumptions on which B depends occur in Γ. Such a tree is a *proof* of B if Γ is empty.

The inference rules of natural deduction can be specified directly in the same manner as those for the L_I sequent system, where the conclusion of the rule corresponds to the head of the clause and the premises to the subgoals. In the next subsection, we discuss this specification. The corresponding notion to cut-free proofs in L_I is *normal* proofs [15] in N_I. To specify N_I so that only normal proofs are built is more complicated than simply adding or removing

clauses, which was all that was required to eliminate the use of cut in sequent proofs. Here, it is a matter of specifying the rules in a different manner. In subsection 4.2, we present such a specification. In subsection 4.3, we discuss alternative ways to specify some of the inference rules retaining the property that only normal proofs are built. The alternatives we present are in fact those needed to be able to merge the N_I specification with the cut-free L_I specification. Finally, in subsection 4.4, we present an alternative way of handling assumptions so that assumptions in N_I correspond to antecedents of sequents in L_I. The result is a specification that can be merged directly with the one for L_I discussed in the previous section.

4.1 A Direct Specification

To represent natural deduction proofs, we introduce the new primitive type `nprf`. Here, the basic proof relation is between proofs and formulas (instead of sequents). We introduce the infix predicate `#` of type `nprf -> form -> o` for this relation. In the previous section, we adopted the convention of using variable names `Q`, `Q1`, `Q2`, etc., to represent sequent proofs. We use `P`, `P1`, `P2`, etc., here for natural deduction proofs. Several of the introduction rules (I-rules) for this system resemble rules that apply to succedents in the sequential system just considered. The $\wedge$-I, $\exists$-I, and $\forall$-I rules correspond to examples given in the previous section and can be specified similarly as follows.

```
(and_i P1 P2) # (A and B) :- P1 # A, P2 # B.
(exists_i T P) # (exists A) :- P # (A T).
(forall_i P) # (forall A) :- pi Y\ ((P Y) # (A Y)).
```

Note that universal quantification is used to handle the proviso on the $\forall$-I rule in the same way that it is used for the $\forall$-R sequent rule.

In natural deduction, unlike sequential systems, we have the additional task of specifying the operation of discharging assumptions. Consider the implication introduction rule in Fig. 2. This rule can very naturally be specified as the definite clause below.

```
(imp_i P) # (A imp B) :- pi W\ ((W # A) => ((P W) # B)).
```

This clause represents the fact that if `P` is a "proof function" which maps an arbitrary proof `W` of formula `A`, to a proof `(P W)` of formula `B`, then `(imp_i P)` is a proof of `(A imp B)`. Here, the proof of an implication is represented by a function from proofs to proofs. The constant `imp_i` has the type `(nprf -> nprf) -> nprf`. Notice that while sequential proofs only contain abstractions of type `tm`, natural deduction proofs may contain abstractions of both types `tm` and `nprf`.

Operationally, when backchaining on this clause, the GENERIC operation is used to choose a new object, say `c`, to replace `W` and play the role of a proof of the formula `A`. The AUGMENT goal is then used to add the assumption `(c # A)` to the current set of program clauses. This clause is then available to use in the

search for a proof of B, *i.e.*, in solving the goal ((P c) # B). The proof of B may contain instances of the proof of A (the constant c). The function P is the abstraction over this constant.

The elimination rules and $\perp$-I can be specified similarly to the introduction rules. For example the following formulas specify the $\wedge$-E and $\exists$-E rules.

```
(and_e1 B P) # A :- P # (A and B).
(and_e2 A P) # B :- P # (A and B).
(exists_e A P1 P2) # B :- P1 # (exists A),
   pi Y\ (pi W\ ((W # (A Y)) => ((P2 Y W) # B))).
```

The $\exists$-E rule contains both a proviso handled by a universal quantifier at the meta-level, and the discharge of an assumption, again handled by meta-level universal quantification and implication. Here, P2 is an abstraction over both the first-order term Y and the proof term W. Note that the proof terms in the heads of these clauses contain more than just the subproofs of the premises as arguments. For example, the missing conjunct in the conclusion of the $\wedge$-E rules is included in the proof terms for these rules.

As for L_I, the set of hh^2 formulas resulting from the direct specification of the rules of N_I serves both as a specification of a theorem prover and as an implementation of a proof checker with respect to the deterministic interpreter. As before, in proof checking, the top-level constant of a proof term completely determines the unique definite clause to be used in backchaining at each step.

4.2 A Specification of N_I That Constructs Normal Proofs

Before discussing specifications of N_I that construct normal proofs, several definitions are required. We begin by defining the notion of normal N_I proofs as in [15]. The main condition for an N_I deduction to be normal is that it must contain no *maximal formula*, that is, a formula that is the conclusion of an I-rule or $\perp_I$ and the major premise of an E-rule, since such applications are redundant (as in the example in Fig. 3 (a)). For the fragment of N_I without the $\vee$ and $\exists$ connectives, this condition is taken as the definition of normal. With these connectives in the logic the condition must be made slightly stronger, and requires some further definitions. A *segment* in a deduction is defined to be a sequence of formula occurrences $A_1, \ldots, A_n$ such that the following hold.

1. A_1 is not the conclusion of an application of $\vee$-E or $\exists$-E.
2. For $i = 1, \ldots, n-1$, A_i is a minor premise of an application of $\vee$-E or $\exists$-E.
3. A_n is not the minor premise of an application of $\vee$-E or $\exists$-E.

All occurrences in a segment are occurrences of the same formula. The deduction in Fig. 3 (b) contains a segment of length 3 of occurrences of $A \wedge B$. As in [15], we will say a segment is the premise of an application of a rule when its last formula is the premise of the rule. A *maximal segment* is a segment that begins with a conclusion of an application of an I-rule or $\perp_I$ and ends with a major premise of

an E-rule. The segment of length 3 in Fig. 3 (b) is in fact maximal. Note that a maximal formula is a special case of a maximal segment. A *normal* deduction is then defined to be a deduction that contains no maximal segment. Figure 3 (c) contains a normal deduction of $r \supset s$ from $\{p \vee q, p \supset (r \supset s), q \supset (r \supset s)\}$.

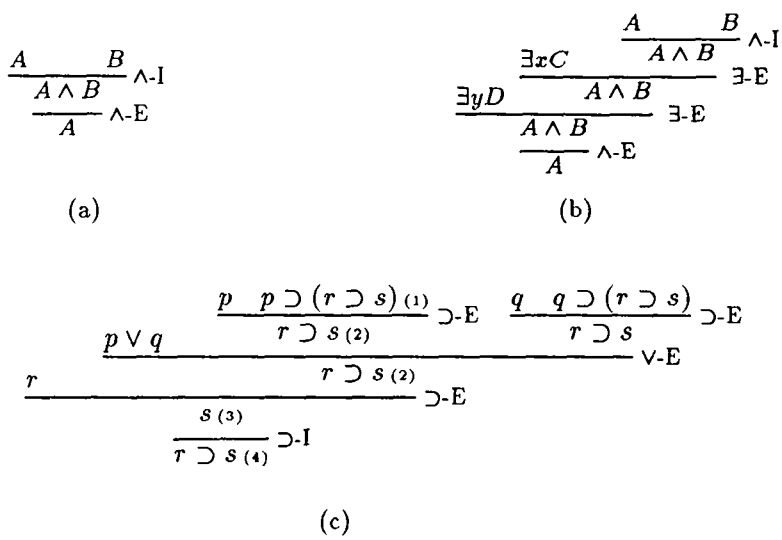

Fig. 3. Maximal formulas, maximal segments, and paths in N_I deductions

Normal deductions can be characterized in terms of the form of certain sequences of formulas called paths. A *path* in a deduction is a sequence of occurrences $A_1, \ldots, A_n$ such that the following conditions hold.

1. A_1 is a leaf that is not discharged by an application of $\vee$-E or $\exists$-E.
2. For $i = 1, \ldots, n-1$, A_i is not the minor premise of an application of $\supset$-E or $\neg$-E. If A_i is the major premise of an application of $\vee$-E or $\exists$-E, then A_{i+1} is an assumption discharged by this application. Otherwise, A_{i+1} is the conclusion of the rule for which A_i is a premise.
3. A_n is either the minor premise of $\supset$-E or $\neg$-E or the root of the deduction.

A path of length 5 (containing 4 segments) in a normal deduction is indicated in Fig. 3 (c). Note that the second segment, indicated by the number (2), has length 2. The other paths in this deduction are:

1. a similar path starting with $q \supset (r \supset s)$
2. $p \vee q, p$
3. $p \vee q, q$
4. r.

We define an *E-path* to be a subsequence of a path such that each segment except the last is a major premise of an E-rule, and an *I-path* to be a subsequence of a path such that each segment except the first is a conclusion of an I-rule or $\perp_I$. We say that a deduction is an *E-deduction* if all paths containing the root are E-paths. Note that the subtree rooted at s in Fig. 3 (c) is an E-deduction.

In a normal deduction, each path contains a series of segments divided into two parts: an E-path followed by an I-path such that the last segment in the E-path called the *minimum segment* is also the first segment in the I-path. In Fig. 3 (c), the formula occurrence s is the minimum segment separating the E-paths and the I-paths of the two paths beginning with $p \supset (r \supset s)$ and $q \supset (r \supset s)$. The minor premises and conclusion of applications of $\vee$-E and $\exists$-E may appear in the E-path or I-path (or both) of all paths through them. In Fig. 3 (c), the middle premise and conclusion of $\vee$-E, for example, occur in the E-path of the indicated path. Note that an E-deduction is normal if and only if all deductions rooted at a minor premise of an application of $\supset$-E or $\neg$-E are normal.

The division of paths in normal deductions into E-paths and I-paths will be reflected in our specification. We will use two relations in specifying the definite clauses for the inference rules. The first one is the `#e` predicate used to relate a formula and a normal E-deduction. For the second, we continue to use `#`, but this time to relate a formula with a normal deduction. Both predicates have type `nprf -> form -> o`. Thus, in a provable formula of the form `(P #e A)`, `P` represents a normal E-deduction of `A`, while in a provable formula of the form `(P # A)`, `P` is a normal deduction of `A`. Operationally, the clauses for the `#e` relation will apply E-rules, and the clauses for the `#` relation will apply I-rules and join I-paths and E-paths at the minimum segment.

Using these two predicates, the introduction rules and $\perp_I$ can be specified exactly as in the direct specification, except that discharged assumptions are added as facts about the `#e` relation since they are one-node E-deductions. They may occur at the leaves in larger deductions and will always occur in the E-paths that pass through them. Thus, the $\supset$-I rule, for instance, is now specified as follows.

```
(imp_i P) # (A imp B) :- pi W\ ((W #e A) => ((P W) # B)).
```

The elimination rules $\wedge$-E, $\supset$-E, $\neg$-E, and $\forall$-E are specified using the `#e` predicate since, in a normal deduction, both the major premise and conclusion of all applications of these rules are roots of normal E-deductions. For example, the clauses for the $\wedge$-E rules now have the following form.

```
(and_e1 B P) #e A :- P #e (A and B).
(and_e2 A P) #e B :- P #e (A and B).
```

The $\supset$-E rule, on the other hand, is represented by the following clause where the subgoal for the minor premise uses the `#` predicate.

```
(imp_e A P1 P2) #e B :- P1 # A, P2 #e (A imp B).
```

This reflects the fact that the subproof at the minor premise can be an arbitrary normal deduction. By the definition of path, the root of this subproof will always be the last formula occurrence in the paths through it. The $\neg$-E rule is similar.

$\vee$-E and $\exists$-E are each specified by two definite clauses, the first corresponding to when the minor premises and conclusion occur in I-paths of paths through them, and the second when they occur in E-paths. The following two clauses are those for the $\vee$-E rule.

```
(or_e A B P P1 P2) # C :- P #e (A or B),
                          pi W\ ((W #e A) => ((P1 W) # C)),
                          pi W\ ((W #e B) => ((P2 W) # C)).

(or_e A B P P1 P2) #e C :- P #e (A or B),
                           pi W\ ((W #e A) => ((P1 W) #e C)),
                           pi W\ ((W #e B) => ((P2 W) #e C)).
```

Note that when the minor premises of this rule are the roots of arbitrary normal deductions, the conclusion is also at the root of an arbitrary normal deduction, as indicated by the use of the `#` relation to relate formula `C` to its various proofs in the head and body of the first of the two clauses above. On the other hand, when the minor premises of this rule are at the roots of normal E-deductions, as in the latter clause above, the conclusion is also the root of a normal E-deduction.

Finally, we must also add the clause `P # A :- P #e A`. This clause serves to join I-paths and E-paths at the minimum segment. Its declarative reading is that any normal E-deduction is a normal deduction.

4.3 Alternative Specifications for Constructing Normal Proofs

We now discuss several modifications that can be made to this specification. This presentation both illustrates alternative ways of specifying the rules of N_I so that only normal proofs get constructed and also brings us closer to a specification that corresponds to cut-free L_I.

In the specification discussed in the previous subsection, the $\neg$-E rule would have the following formulation, since it is similar to $\supset$-E.

```
(neg_e A P1 P2) #e false :- P1 # A, P2 #e (neg A).
```

In a normal N_I deduction, any occurrence of $\perp$ is always in the minimum segment of paths through it, since it cannot be the major premise of an E-rule or the conclusion of an I-rule. As a result, we have the option of modifying the above formula so that `#` instead of `#e` appears in the head of the clause. With this modification, it is still the case that `P` in a provable formula of the form `(P #e A)` represents an E-deduction of `A`, but now it is one that does not end in $\neg$-E. Operationally, the clauses for the `#` relation now build I-paths in a goal directed fashion, and possibly also add the last segment in the E-paths of paths through `false`. Clauses for `#e` will then add the remaining E-paths.

We can simplify the specification of the previous subsection if we consider the following refinement of normal deductions. We define an *E-segment* in a

normal deduction to be a segment whose last occurrence is the conclusion of an application of $\vee$-E or $\exists$-E and the major premise of an E-rule. Note that maximal segments are a special case of E-segments. As pointed out to Prawitz by Martin-Löf [14], the definition of normal can be sharpened to require that normal deductions contain no E-segments or maximal formulas. We call such deductions *S-normal* deductions. In an S-normal deduction, every minor premise or conclusion of an application of $\vee$-E and $\exists$-E will appear either in the minimum segment (in which case it is both in the I-path and E-path) or only in the I-path of paths through it. The deduction below is a modification of the deduction in Fig. 3 (c) that meets the extra restriction on segments to make it S-normal.

$$\dfrac{p \vee q \qquad \dfrac{r \qquad \dfrac{p \quad p \supset (r \supset s)}{r \supset s}\supset\text{-E}}{s}\supset\text{-I}}{r \supset s}\vee\text{-E} \qquad\qquad \dfrac{\dfrac{r \qquad \dfrac{q \quad q \supset (r \supset s)}{r \supset s}\supset\text{-E}}{s}\supset\text{-I}}{r \supset s}$$

The specification for N_I that we will eventually merge with the specification for L_I is one that builds deductions in this sharpened normal form. In order to discuss this specification we refine some definitions given earlier. By an *E-path*, we will still mean a subsequence of a path such that each segment except the last is a major premise of an E-rule, but now we add the additional restriction that each segment must be of length 1. The definition of I-path remains the same. In an E-deduction, all paths containing the root must now meet the additional restriction on E-paths. With respect to this modified definition, each path in an S-normal deduction will still contain an E-path followed by an I-path, but now there may not be any overlap in these two parts. The minimum segment will always be in the I-path, but will not be in the E-path if it has length greater than 1.

Note that under these new definitions, the latter of the two clauses given for $\vee$-E in the previous subsection is no longer correct. Even when all three premises are E-deductions, the conclusion will not be, since the root will be in a segment of length greater than 1, and thus paths through it will not be E-paths. Hence, we must eliminate this clause and the similar one for $\exists$-E. The simple elimination of these two clauses is all that is required to obtain a specification that constructs only S-normal deductions. In the remainder of this paper, we use the term "normal" to mean "S-normal" since we now only consider deductions in S-normal form.

With the above modifications, the remaining clauses with **#e** in the head are those for $\wedge$-E, $\supset$-E, and $\forall$-E. These rules can also be modified. We illustrate using the $\wedge$-E rules. The clauses in subsection 4.2 specifying the $\wedge$-E rules can be replaced by the following clause with **#** in the head.

```
PC # C :- P #e (A and B),
          (((and_e1 B P) #e A) => (((and_e2 A P) #e B) => (PC # C))).
```

The declarative reading of this clause is as follows: PC is a proof of formula

C if there is a normal E-deduction of conjunct (A and B) (whose proof is P), and from the assumptions that A and B are provable separately (with proofs (and_e1 B P) and (and_e2 A P) respectively) it can be shown that PC is a proof of C. Operationally, in attempting to find a normal deduction for any formula C, this clause will look for an E-deduction of a conjunction (A and B), apply both versions of the $\wedge$-E rule to it to obtain two new E-deductions, add the new subproofs as atomic program clauses, and attempt to find a normal deduction for C in the environment extended with these new assumptions.

As stated and proved in [15], normal deductions have the *subformula property*, that is, every formula occurring in a normal deduction of A from Γ is a subformula of A or of some formula in Γ. In fact, every formula occurring in an E-path is a subformula of a formula in Γ, every formula occurring in an I-path is a subformula of A, and every formula occurring in a minimum segment is a subformula of both A and some formula in Γ. This property is reflected in the following operational description of our new specification. The clauses for the I-rules apply the rules in a backward direction so that the formulas in the subgoals (which correspond to the premises) are always subformulas of the formula in the head of the clause (the conclusion). In contrast, the clauses for $\wedge$-E, $\supset$-E, and $\forall$-E apply the rules in a forward direction from the assumptions so that the conclusion is always a subformula of the major premise. In applying E-rules, new E-deductions get built from existing E-deductions and are then added to the program as new facts about the #e relation.

Note that this program can no longer serve as a proof checker with respect to the deterministic interpreter. It is no longer the case that the top-level constant of a proof term completely determines the definite clause to be used in backchaining. In fact, clauses like the specification of $\wedge$-E above can always be used in attempting to prove an atomic goal for the # predicate since any formula and proof term will be instances of the formula and proof in the head of the clause.

4.4 Explicit vs. Implicit Representation of Assumptions in Natural Deduction

In specifying N_I we showed that it was quite natural to specify the discharge of assumptions "implicitly" using universal quantification and implication at the meta-level. It is also possible to explicitly keep track of assumptions by storing them in a list, and make use of a membership predicate to extract individual elements in much the same way that antecedents were handled in the specification of L_I. For N_I, such assumption lists will contain pairs of formulas associated with their E-deductions. We will call such lists of pairs *contexts*. We can obtain an "explicit context" specification of N_I via a systematic modification of the definite clauses of any of the "implicit context" specifications discussed so far. We illustrate using the formulation in the previous subsection.

We continue to use #e and # for the basic relations between a formula and its deductions, but no longer as predicates; they will now have the type nprf

`-> form -> judg` where `judg` is a new primitive type introduced to represent these basic judgments. Contexts will be lists of elements of type `judg_l` where `judg_l` is, of course, the primitive type introduced for lists of elements of type `judg`. We must modify each definite clause so that it has an extra argument for a context, which corresponds to the set of assumptions that exist at the time the rule is applied. We use the sequent arrow `-->` as our predicate, now used to form "judgment sequents." This predicate has type `judg_l -> judg -> o` and expresses the relation between a context and a single formula paired with a normal deduction or normal E-deduction.

For those clauses that do not involve the use of implication to add assumptions to the program, the modification simply requires adding a list and sequent arrow to form a judgment sequent in the head and subgoals of each clause. For example, the clause for $\wedge$-I is rewritten as follows.

```
Gamma --> ((and_i P1 P2) # (A and B)) :- Gamma --> (P1 # A),
                                        Gamma --> (P2 # B).
```

The discharge of assumptions as in the $\supset$-I rule is specified as below where the new assumption gets added to the context rather than the program.

```
Gamma --> ((imp_i P) # (A imp B)) :-
   pi W\ (((W #e A)::Gamma) --> ((P W) # B)).
```

All other formulas can be modified similarly. For example, the clause for $\wedge$-E adds two assumptions to the context.

```
Gamma --> (PC # C) :- Gamma --> (P #e (A and B)),
   (((and_e1 B P) #e A)::((and_e2 B P) #e B)::Gamma) --> (PC # C).
```

The formula expressing the fact that a normal E-deduction is a normal deduction is replaced by the following clause.

```
Gamma --> (P # A) :- Gamma --> (P #e A).
```

As before, this clause operationally joins E-paths and I-paths at the minimum segment. Finally, we need the following clause for completing E-deductions.

```
Gamma --> (P #e A) :- memb (P #e A) Gamma.
```

Of course, the `memb` predicate here is at type `judg`. In previous implicit context specifications, we did not need an explicit clause for closing E-deductions. There, an E-deduction was closed by unifying an atomic goal formula of the form `(P #e A)` with an atomic clause of the program.

This completes the description of the systematic modification from implicit to explicit context specifications. We now discuss two final modifications that can be made to this particular specification of N_I.

Recall that in the specification in the previous subsection, all clauses have `#` as the predicate occurring in the head. Thus in the modified specification here, all clauses except for the one immediately above have `#` as the relation on the

right of the sequent arrow in the head of the clause. Hence, whenever an atomic formula of the form `(Gamma --> (P #e A))` succeeds, it must be the case that `(P #e A)` is in the context `Gamma`. We could in fact remove the clause above for closing E-deductions, and replace each subformula of the form `(Gamma --> (P #e A))` with `(memb (P #e A) Gamma)` directly in the formulas specifying the inference rules. The formula for $\wedge$-E would then be written as follows.

```
Gamma --> (PC # C) :- memb (P #e (A and B)) Gamma,
   (((and_e1 B P) #e A)::((and_e2 A P) #e B)::Gamma) --> (PC # C).
```

After making such a modification, the resulting set of formulas is such that only the `#` relation appears on the right, while only the `#e` relation appears on the left of the sequent arrow. Thus, we can distinguish normal E-deductions from arbitrary normal deductions simply by where they occur in judgment sequents. As a result, we no longer need two distinct relations. They can be merged into one. We do so here by simply eliminating the `#e` relation and adopting `#` as the single relation between a formula and a deduction. Using this formulation, the above clause for $\wedge$-E becomes the following formula.

```
Gamma --> (PC # C) :- memb (P # (A and B)) Gamma,
   (((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> (PC # C).
```

The specification obtained by making similar modifications to all of the clauses is the one we will now directly merge with the L_I specification.

5 Transforming L_I Proofs to N_I Proofs

The first step in merging the specifications of L_I and N_I is to combine the data structures for sequents and judgments. We do so using the same constants as before, but now give them the following types.

```
type    #       nprf -> form -> judg.
type    -->     judg_l -> judg -> seq.
type    >-      lprf -> seq -> o.
```

As before `#` is used for the relation between a formula and N_I deduction. We also again use the sequent arrow for judgment sequents. Here the type `seq` replaces `o` since it will no longer serve as the top-level predicate. As in the specification of L_I, `>-` is the top-level predicate with the same type as before. Note, though, that the second argument is a judgment sequent in this case. An atomic formula now has the form `(Q >- (Gamma --> (P # A)))` where `Gamma` is a list of judgment pairs. If such a formula is provable, it will be the case that `Q` represents a cut-free L_I proof of the sequent whose antecedent contains the formulas in `Gamma` and whose succedent is `A`, and `P` represents a normal N_I deduction of `A` from the formulas in `Gamma`.

In the clauses in this specification, the L_I rules for introducing a connective on the right of the sequent will be simultaneously specified with the corresponding N_I introduction rule. Similarly, the rules for introducing a connective on the

left in L_I will be specified with the corresponding elimination rule. Finally, the $\perp$-R rule in L_I and the $\perp_I$ rule in N_I will be specified together. For example, the first formula below simultaneously specifies the $\wedge$-R and $\wedge$-I rules from L_I and N_I, respectively, while the second formula specifies both $\wedge$-L and $\wedge$-E.

```
(and_r Q1 Q2) >- (Gamma --> ((and_i P1 P2) # (A and B))) :-
   Q1 >- (Gamma --> (P1 # A)), Q2 >- (Gamma --> (P2 # B)).

(and_l (A and B) Q) >- (Gamma --> (PC # C)) :-
   memb (P # (A and B)) Gamma,
   Q >- ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> (PC # C)).
```

They are obtained by a straightforward merging of the corresponding formulas of the separate specifications. They illustrate how N_I proof terms are associated with formulas within a sequent on the right and left, respectively, while the L_I proof terms are associated with the entire sequent using the top-level relation.

The following clause simultaneously specifies $\forall$-R and $\forall$-I.

```
(forall_r Q) >- (Gamma --> ((forall_i P) # (forall A))) :-
   pi Y\ ((Q Y) >- (Gamma --> ((P Y) # (A Y)))).
```

This clause illustrates how universal quantification at the meta-level is used to handle simultaneously the provisos on both $\forall$-R and $\forall$-I. Note that both `Q` and `P` are abstractions over the variable `Y`.

As a final example, consider the formula below for $\supset$-R and $\supset$-I.

```
(imp_r Q) >- (Gamma --> ((imp_i P) # (A imp B))) :-
   pi W\ (Q >- (((W # A)::Gamma) --> ((P W) # B))).
```

Universal quantification is used to introduce a constant to replace the variable `W` and serve as a proof for hypothesis `A`. As before, the term `P` is an abstraction over this constant. It represents a function from proofs of `A` to proofs of `B`. Note, on the other hand that, as in the L_I specification, `Q` is not an abstraction.

The complete specification for L_I and N_I is given in Fig. 4. Operationally, this program can take on several roles. In a query of the form:

```
(Q >- (nil --> (P # A)))
```

where both `Q` and `P` are logic variables, the program behaves as a theorem prover, and simultaneously constructs proofs in both inference systems. If `P` is also specified, then the program acts as a proof transformer, transforming an N_I proof `P` to an L_I proof `Q`. Conversely, an L_I proof `Q` can be used to guide the construction of an N_I proof `P`. The program is complete with respect to the deterministic depth-first interpreter for this latter transformation, for the same reason that the L_I specification is complete as a proof checker. The constant at the head of the proof term `Q` uniquely determines which definite clause must be used at each step. The converse, however, is not true since as discussed in subsection 4.3, this formulation of the N_I rules cannot serve as a proof checker with respect to the deterministic interpreter because of the clauses for the $\wedge$-E,

```
(initial A) >- (Gamma --> (P # A)) :- memb (P # A) Gamma.

(and_r Q1 Q2) >- (Gamma --> ((and_i P1 P2) # (A and B))) :-
   Q1 >- (Gamma --> (P1 # A)), Q2 >- (Gamma --> (P2 # B)).

(or_r1 Q) >- (Gamma --> ((or_i1 P) # (A or B))) :-
   Q >- (Gamma --> (P # A)).

(or_r2 Q) >- (Gamma --> ((or_i2 P) # (A or B))) :-
   Q >- (Gamma --> (P # B)).

(imp_r Q) >- (Gamma --> ((imp_i P) # (A imp B))) :-
   pi W\ (Q >- (((W # A)::Gamma) --> ((P W) # B))).

(neg_r Q) >- (Gamma --> ((neg_i P) # (neg A))) :-
   pi W\ (Q >- (((W # A)::Gamma) --> ((P W) # false))).

(forall_r Q) >- (Gamma --> ((forall_i P) # (forall A))) :-
   pi Y\ ((Q Y) >- (Gamma --> ((P Y) # (A Y)))).

(exists_r T Q) >- (Gamma --> ((exists_i T P) # (exists A))) :-
   Q >- (Gamma --> (P # (A T))).

(false_r Q) >- (Gamma --> ((false_i P) # A)) :-
   Q >- (Gamma --> (P # false)).

(and_l (A and B) Q) >- (Gamma --> (PC # C)) :-
   memb (P # (A and B)) Gamma,
   Q >- ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> (PC # C)).

(imp_l (A imp B) Q1 Q2) >- (Gamma --> (PC # C)) :-
   memb (P2 # (A imp B)) Gamma, Q1 >- (Gamma --> (P1 # A)),
   Q2 >- ((((imp_e A P1 P2) # B)::Gamma) --> (PC # C)).

(forall_l (forall A) T Q) >- (Gamma --> (PC # C)) :-
   memb (P # (forall A)) Gamma,
   Q >- ((((forall_e A T P) # (A T))::Gamma) --> (PC # C)).

(neg_l (neg A) Q) >- (Gamma --> ((neg_e A P1 P2) # false)) :-
   memb (P2 # (neg A)) Gamma, Q >- (Gamma --> (P1 # A)).

(or_l (A or B) Q1 Q2) >- (Gamma --> ((or_e A B P P1 P2) # C)) :-
   memb (P # (A or B)) Gamma,
   pi W\ (Q1 >- (((W # A)::Gamma) --> ((P1 W) # C))),
   pi W\ (Q2 >- (((W # B)::Gamma) --> ((P2 W) # C))).

(exists_l (exists A) Q) >- (Gamma --> ((exists_e A P1 P2) # B)) :-
   memb (P1 # (exists A)) Gamma,
   pi Y\ (pi W\ ((Q Y) >- (((W # (A Y))::Gamma) --> ((P2 Y W) # B)))).
```

Fig. 4. Definite clauses for simultaneous specification of L_I and N_I

$\supset$-E, and $\forall$-E rules. If however, the depth-first search strategy of the interpreter were replaced by breadth-first search, the transformation in this direction would be achieved by this program.

Alternatively, we could modify the specification in Fig. 4 so that the transformation from N_I to L_I proofs works with respect to the depth-first interpreter. One simple approach involves taking into account the form of E-deductions in S-normal proofs. In such deductions, there is one path to the root that begins at an assumption and contains a sequence of applications of $\wedge$-E, $\supset$-E, $\forall$-E, possibly ending in an application of $\neg$-E. One or more of the sequent proofs to which such an E-deduction is related will contain a sequence of $\wedge$-L, $\supset$-L, and $\forall$-L rules occurring in reverse order to the corresponding I-rules in the natural deduction proof. It is straightforward to write a program to convert an E-deduction to such a corresponding "inverted" sequent proof. Then, a program to convert general N_I normal deductions to L_I proofs can be obtained by removing the clauses for the $\wedge$-E, $\supset$-E, and $\forall$-E rules from the clauses in Fig. 4 and modifying the remaining program to make use of such an inversion procedure for E-deductions.

6 Conclusion

The program we have presented provides both a declarative and an operational description of the correspondence between the sequent calculus and natural deduction for intuitionistic logic. An interesting next step would be to extend this approach to provide illustrations of other well-known proof-theoretical results. One obvious candidate is the correspondence between cut-elimination in sequent systems and proof normalization in natural deduction. Such a specification may provide a program that simultaneously performs both operations.

In Sect. 4.4, we illustrated how to convert the specification of Sect. 4.3, which uses meta-level implication for the discharge of assumptions, into a specification using explicit assumption lists. It is easy to see that this operation can be performed on any of the "implicit context" specifications of natural deduction presented here. In fact, this operation is not limited to an intuitionistic object-logic. Specifications of natural deduction for classical logic or higher-order logic, for example, could be similarly transformed. Any such explicit context specification can be viewed as a specification of a sequent style inference system for the same logic. In the case of N_I, the fact that the inference rules could be specified in such a way that the corresponding sequent system was exactly L_I without the cut rule is a result of the close correspondence between the two. The well-known sequent and natural deduction inference systems for classical logic, for example, cannot be so easily related in this way.

Acknowledgements

I am grateful to Dale Miller and Peter Schroeder-Heister for valuable discussions and comments related to this paper. This research was supported in part by

grants ARO-DAA29-84-9-0027, ONR N00014-88-K-0633, NSF CCR-87-05596, DARPA N00014-85-K-0018, and ESPRIT Basic Research Action 3245.

References

1. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
2. Michael Dummett. *Elements of Intuitionism.* Clarendon Press, Oxford, 1977.
3. Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. May 1989.
4. Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, University of Pennsylvania, August 1989.
5. Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, Argonne Ill., May 1988.
6. Gerhard Gentzen. Investigations into logical deductions, 1935. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland Publishing Co., Amsterdam, 1969.
7. J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus.* Cambridge University Press, 1986.
8. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
9. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. To appear in the *Annals of Pure and Applied Logic.*
10. Gopalan Nadathur and Dale Miller. Higher-order horn clauses. April 1988. To appear in the *Journal of the ACM.*
11. Gopalan Nadathur and Dale Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*, MIT Press, 1988.
12. Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, September 1989.
13. Garrel Pottinger. Normalization as a homomorphic image of cut-elimination. *Annals of Mathematical Logic*, 12(3):223–357, 1977.
14. Dag Prawitz. Ideas and results in proof theory. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 235–307, North-Holland, Amsterdam, 1971.
15. Dag Prawitz. *Natural Deduction.* Almqvist & Wiksell, Uppsala, 1965.
16. J. I. Zucker. Cut-elimination and normalization. *Annals of Mathematical Logic*, 1(1):1–112, 1974.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399